

**Utility Patent Application**

**For**

**DYNAMIC RECOMPILER**

**By**

**Randal N. Linden**

**Related Application**

This is a continuation of Provisional Application No. 60/175,008, filed on January 7, 2000.

**BACKGROUND OF THE INVENTION**

**1. Field of the Invention**

The present invention relates to a method and apparatus for emulating source instructions on a target information processing system and, more particularly, to a scheme for translating source instructions to target instructions executable on the target information processing system.

**2. Description of Related Art**

Information processing system or computing systems comes in many configurations. They include without limitation the generally known computers (such as personal

computers), personal digital assistant, video game consoles, application specific control systems, and other systems and devices that incorporate a processing unit. For example, computers have general-purpose central processing units (CPU) which are designed to execute a specific set of instructions.

5       A CPU of one family, such as the Motorola 680X0 family of processors, manufactured by Motorola, Inc., Phoenix, Ariz., executes instructions unique to this family, a CPU of another family, such as the Intel 80X86 manufactured by Intel Corp., Sunnyvale, Calif., executes instructions unique to that family, and a PowerPC processor family, used in a Macintosh computer manufactured by Apple Computer, Inc.,  
10   Cupertino, Calif., executes instructions specific to that family. These instructions comprise part of the operating systems of the respective computers in which the respective CPUs reside. Typically, application software programs are then written to run on the respective operating systems.

As another example, video game manufacturers, such as Sony, developed a game  
15   player based on a proprietary hardware (processing) platform that runs only game software in a CD format which is specifically developed for its platform. Sony game software would not run on other hardware platforms. Game software based on other platforms would not run on Sony players.

A computer manufacturer will design a computer system with a given CPU and will  
20   want to maximize market penetration by having more rather than fewer applications software being able to run on its system. For example, as indicated above, Apple Computer manufactures the PowerPC-based Macintosh line of computers. Applications software that has been written to run on operating systems for the Motorola 680X0

family, for example, may not all run on the PowerPC-based machines. Accordingly, a field of technology has developed in which a given computer having one type of CPU, called a target, will include an emulator that allows the target computer to emulate the instructions, called the source, of another type of CPU. Thus, the target computer will  
5 have stored in memory source instructions that may be called in response to applications software, target instructions emulating the source instructions and executable by the target CPU, and an emulator that causes one or more target instructions to be executed in response to a given source instruction. Thus, the given computer can execute target instructions of its own machine, and through emulation execute source instructions.

10 Two main types of emulation strategies currently are available in the emulation field. The first strategy is known as "interpretation", in which each source instruction is decoded in turn as it is addressed, causing a small sequence of target instructions then to be executed that emulate the source instruction. The main component of an emulator is typically an interpreter that converts each instruction of any program in machine  
15 language A into a set of instructions in machine language B, where machine language B is the code language of the target computer on which the emulator is being used. In some instances, interpreters have been implemented in computer hardware or firmware, thereby enabling relatively fast execution of the emulated programs.

20 The overhead, in terms of speed, for decoding each source instruction each time the source instruction is called is relatively small, but present for each instruction. Consequently, a given source instruction that is addressed and, hence, decoded, many times in the course of running a program will slow the execution time of the overall program being run, i.e., overhead will increase.

The other main emulation strategy is known as "translation", in which the source instructions are analyzed and decoded. This is also referred to as "recompilation" or "cross-compilation". It is well known that the execution speed of computer programs is dramatically reduced by interpreters. It is not uncommon for a computer program to run  
5 ten to twenty times slower when it is executed via emulation than when the equivalent program is recompiled into target machine code and the target code version is executed. Due to the well known slowness of software emulation, a number of products have successfully improved on the speed of executing source applications by dynamically cross-compiling portions of such program at run time into target machine code, and then  
10 executing the recompiled program portions. While the cross-compilation process typically takes 50 to 100 machine or clock cycles per instruction of the source, the greater speed of the resulting target machine code is, on average, enough to improve the overall speed of execution of most source applications.

The primary reason that overall execution speed is improved by cross-compilation is  
15 that most programs contain execution loops of instructions that are repeatedly executed hundreds, thousands, or even millions of times during a typical execution of the program. The source instructions are analyzed and decoded only once, i.e., the first time they are addressed, and the target instruction stream is generated and stored in memory, usually a RAM memory that may be a cache memory. By avoiding repeated  
20 interpretation of the instructions in such loops, substantial execution time is saved. Consequently, subsequent emulation of the same source instruction may be performed quickly because the decoding overhead is nonexistent.

While run time cross-compilation of source applications is well known to those skilled in the art, there are several areas in which existing cross-compilation systems have fallen short of their potential. For example, it is necessary to have a relatively large buffer or cache memory in the target computer. A block of memory in the target computer's memory address space is set aside to store the target instruction stream generated during the initial translation process. If this block is large enough to contain the entire translated target instructions, emulation will proceed at the maximum rate since, in response to a given source instruction, the RAM memory can be quickly addressed to access the corresponding sequence of target instructions for execution.

Otherwise, there is translation overhead necessary to handle translation of the program in small sections during the execution of the source program. However, it often is expensive or prohibitive to set aside a large enough RAM memory. Furthermore, it may also be impossible to determine how large a RAM memory must be allocated in the target computer's memory address space to contain the translated program. Competition for RAM or cache memory space with other application software running on the same target machine may further limit the execution speed of source applications. For applications involving video graphics, it is important that continuity in the graphics is maintained by the emulation of the source application.

Consequently, a competing interest exists between the size of the RAM memory and the translation overhead. If the RAM memory is relatively large, the translation overhead will be relatively low, but at the high cost of memory. If the RAM is small, reducing memory cost, the translation overhead may be high due, for example, to continually writing newly translated code to the RAM. These drawbacks are more significant with

low-end machines running on a slower processor and running on limited amounts of memory.

Accordingly, it is desirable to provide an improved system and method for dynamic recompilation that produces a more efficient set of translated code and uses less memory.

## SUMMARY OF THE INVENTION

The present invention is directed to an improved system and method for dynamic recompilation of source software for execution on a target information processing system.

5 The present invention overcome the short comings of the prior art by optimizing the translated code to improve execution speed and to require less memory overhead, thereby reducing the translation overhead and further improving execution speed of the translated code.

10 In accordance with one aspect of the present invention, in making the translation, the dynamic recompiler considers not only the specific instructions of the source software, but also the intent and purpose of the instructions, to translate to a set of equivalent code for the target system that is optimized based on the target processor where the translated instructions will be running on. The dynamic recompiler essentially determines what the source operation code is trying to accomplish and the best way of doing it at the target  
15 processor, in an “interpolative” and context sensitive fashion. The source instructions are processed in blocks of varying sizes by the dynamic recompiler. By processing the block of instruction en masse, the dynamic recompiler considers the instructions that come before and after a current instruction so as to be able to select the most efficient approach out of several available approaches for encoding the operation code for the target  
20 processor to perform the equivalent tasks specified by the source instructions.

In one embodiment of the present invention, the dynamic compiler comprises three stages: (1) a decoding stage for decoding the source instructions and parameters, creating an instruction stream that is optimized based on the source instructions and parameters;

(2) an optimization stage for optimizing the flow of information and related operation code based on the characteristics of the target processor; and (3) an encoding stage for encoding instructions specifically for the target processor to achieve the intended results, including further optimizing the operation code for the target processor.

- 5 For purpose of illustrating the inventive concept, the present invention is described using the example of recompilation of video game software for playing on a personal computer system.

## **BRIEF DESCRIPTION OF THE DRAWINGS**

Fig. 1 a schematic representation of one embodiment of a computer system that executes the dynamic recompilation process in accordance with the present invention.

5 Fig. 2 is a schematic representation of the stages in one embodiment of the dynamic recompiler of the present invention.

Fig. 3 is a schematic flow diagram of the dynamic recompilation process in accordance with one embodiment of the present invention.

## DESCRIPTION OF THE ILLUSTRATED EMBODIMENTS

The present description is of the best presently contemplated mode of carrying out the invention. This description is made for the purpose of illustrating the general principles  
5 of the invention and should not be taken in a limiting sense. The scope of the invention is best determined by reference to the appended claims.

The present invention is directed to emulation of source instructions on a target information processing system. To facilitate an understanding of the principles and features of the present invention, they are explained herein below with reference to its  
10 deployments and implementations in illustrative embodiments. By way of example and not limitation, the present invention is described herein-below in reference to examples of deployments and implementations for translating software written for a proprietary game player for playing on a personal computer.

The present invention can find utility in a variety of implementations without  
15 departing from the scope and spirit of the invention, as will be apparent from an understanding of the principles that underlie the invention. It is understood that the dynamic recompilation concept of the present invention may be applied to recompilation of software designed for other hardware and/or software platforms, whether in an information exchange network environment or otherwise. For example, the present  
20 invention may be applied to dynamically recompile instructions based on the Motorola 680X0 platform to instructions to run on the Intel 80X86 platform, for game, business, productivity and other types of application software.

It will be appreciated that the line between hardware and software is not always sharp, it being understood by those skilled in the art that such networks and communications facility involve both software and hardware aspects. A method or process is here, and generally, conceived to be a self-consistent sequence of steps leading  
5 to a desired result. These steps require physical manipulations of physical quantities.

Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the  
10 like. It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Useful devices for performing the operations of the present invention include, but are not limited to, general or specific purpose digital processing and/or information  
15 processing devices, which devices may be standalone devices or part of a larger system.

As used in the context of the present invention, and generally, digital processing and information processing systems may include computers (such as personal computers), personal digital assistant, video game consoles, application specific systems, and other systems and devices that incorporates a processing unit. The devices may be selectively  
20 activated or reconfigured by a program, routine and/or a sequence of instructions and/or logic stored in the devices. In short, use of the methods described and suggested herein is not limited to a particular processing configuration. Prior to discussing details of the

inventive aspects of the present invention, it is helpful to discuss one example of an information processing system in which the present invention may be implemented.

### Target Processing System

5

Fig. 1 schematically illustrates one embodiment of a computer system 20 that incorporates dynamic recompilation feature of the present invention. The computer system 20 includes a processor 22 (e.g., an Intel 80X86 processor), internal random-access memory ("RAM") 23 and read-only memory ("ROM") 25, and a data bus architecture for coupling the processor 22 to various internal and external components. A mass storage device 34, such as a hard disk drive, is coupled to the processor 22 for storing utility and application software (including the dynamic recompiler of the present invention) and other data. The application software is executed or performed by the processor 22. A data read/write device 37, such as a CD-ROM drive, DVD drive or floppy disk drive, is provided. The source instructions may be stored in the hard drive 34 or on a CD-ROM for playbacks using the CD-ROM drive 37.

User actuable input devices are also coupled to the processor 22, including a cursor positioning device 30 and a keyboard 32 in accordance with the present invention. The cursor positioning device 30 is representative of any number of input devices that produce signals corresponding to a cursor location on the display 24, and may include by way of example, a game control console, a joy-stick, a mouse, a trackball, an electronic pen, or a touch-pad, which may be an integral part of the keyboard 32. A display 24 is coupled to the processor 22 through a video controller 28. The video controller 28

coordinates the presentation of information on the display 24 in one or more windows 26.

Generally, the windows 26 are scalable, thus permitting a user to define the size and location of a particular window 26 on the display 24.

## 5 Dynamic Recompilation

By way of example and not limitation, the dynamic recompilation concept of the present invention is discussed in reference to the example of emulation of proprietary video game software for playing on a personal computer system. More specifically, the present invention is discussed in reference to the emulation of video game software that has been written to run on a MIPS R-3000 processor based system (source; e.g. Sony PlayStation) for running on a Intel 80X86 processor based system (target).

In accordance with one aspect of the present invention, in making the translation, the dynamic recompiler considers not only the specific instructions of the source software, but also the intent and purpose of the instructions, to translate to a set of equivalent code for the target system that is optimized based on the target processor where that the translated instructions will be running on. The dynamic recompiler essentially determines what the source operation code is trying to accomplish and the best way of doing it at the target processor, in an "interpolative" fashion. The source instructions are processed in blocks of varying size by the dynamic recompiler at one time. By processing the block of instruction en masse, the dynamic recompiler considers the instructions that come before and after a current instruction so as to be able to select the most efficient approach out of several available approaches for encoding the operation

code for the target processor to perform the equivalent tasks specified by the source instructions.

In one embodiment of the present invention as schematically illustrated in Fig. 2, the dynamic compiler 10 comprises three stages: (1) a decoding stage for decoding the source instructions and parameters, creating an instruction stream that is optimized based on the source instructions and parameters; (2) an optimization stage 14 for optimizing the flow of information and related operation code based on the characteristics of the target processor; and (3) an encoding stage 16 for encoding instructions specifically for the target processor to achieve the intended results, including optimizing the operation code for the target processor.

Fig. 3 is a schematic flow diagram of the dynamic recompilation process in accordance with one embodiment of the present invention.

#### Decoding Stage

Referring to Fig. 3, at the decoding stage 12, the source instructions are fetched at 40, e.g. from a CD-ROM read by a CD-ROM drive, a hard drive, or in real time streamed over a network. The instructions are analyzed and decoded at 42 in blocks of varying sizes that depend on several factors, such as the location of the instruction and data in memory or register, the available system resource (e.g., registers, memory), timing, instruction sequences based on functionality of the instructions (e.g., entry point), target processor's operational characteristics. In essence, the purpose of the instructions and flow of information (e.g., where data comes from and going to) are determined at 42.

The decoding stage goes through the block of source instructions and analyzes, examines the operation codes, and outputs an instruction string which is basically a breakdown of what the functional operations are doing. For example for addition, it determines what is the addition from, which two numbers are being added, where the numbers come from, and where to put the result of the addition when done. The instruction string output stream contains information on the flow of information in the program.

Taking the decoded instruction, a preliminary optimization of the instruction stream is conducted at 44 to include the following optimization tasks: (a) ignore no-op codes; (b) reduce parameters that do not affect the operation, e.g., Register 3 = Register 1 OR Register 1 (the parameter "OR Register" is removed), Register 3 = Register 1 +0 (the "+0" parameter is removed); (3) translate the instruction sequence into another sequence where the result is always a constant; e.g., Register 3 = Register 1 XOR Register 1 (translate to Register 3=0); and (d) ignore the entire instruction if it has no effect; e.g., Register 3 = Register 3 OR Register 3 (ignore, since the original value of Register 3 is maintained). The preliminary optimization step 44 outputs the optimized instruction stream in preparation for subsequent flow optimization and encoding.

The decoding stage may be further configured to analyze the instruction stream for data flow to determine if the operation is overridden by subsequent operations. For example, consider the following sequence of instructions  $C=A+B$  and  $C=M+Q$ , whereby there is no intermediate instruction involving C. The decoding stage determines that the value of  $C=A+B$  would be overridden by the value of  $C=M+Q$ , therefore the decoding stage would optimize the decoded instruction stream by omitting the  $C=A+B$  instruction.

Optimization Stage

The optimization stage 14 takes the instruction stream from the decoding stage 12 and optimizes the flow of information and related operation code based on the characteristics of the target processor. It basically analyzes the instruction stream that was created by the decoding stage, and determines the order and sequence of the operation code for the target processor. While the instruction stream based on the source processor might be addition, subtraction, etc., the instruction stream optimized for the target processor is not instructions per se. Instead, the optimized instruction stream describes the flow of information that is efficient for the target processor. It is noted that one of the core differences in processors is the way they deal with information and the flow of information. At the conclusion of the optimization stage 14, the output is an instruction stream that contains information on the equivalent flow of information to be handled by the target processor to emulate the intended results of the source instructions. One can look at this optimization process as an interpolation from the source instructions to the equivalent results to be achieved by the target processor, independent of the operations specified by the source instructions, but dependent on the intended purpose and flow of the source instructions and how they are to be handled by the target processor to achieve an equivalent result.

To illustrate the optimization stage 14, consider the difference between the MIPS R-3000 processor and the Intel 80X86 processor. The R-3000 processor has 32 general-purpose registers, to which the processor has instantaneous access. There is practically no time delay in accessing the registers as compared to RAM memory. Data retrieval

from memory requires a certain amount of time (e.g., 16 nanoseconds). Consider the addition of two numbers residing in memory. The first number is fetched from memory and placed into one of the registers, and the other number is then fetched from memory and placed into another register. The registers are summed to complete the addition

5 function, and the result is placed back into memory. Since the fetching from memory is relatively slow, the optimization stage 14 would minimize the number of fetches from memory that the target processor has to undertake. For example, consider adding the numbers A, B, and C separately to D. The long way of doing this for the source processor would be to fetch the numbers from memory required for the respective

10 additions separately to complete the task; i.e., the number D is fetched three times for the three separate additions. Using the optimization scheme of the present invention for a target Intel 80X86 processor, the numbers A, B, C and D would be fetched from memory and placed in registers. Then D is added to A, B and C separately, and the new values are stored into memory. Accordingly, D is fetched once and used three times, instead of  
15 being fetched three times. This reduced instruction set and memory access is especially useful for the 80X86 processor, because it has only 7 registers.

As a further example, the optimization stage 14 efficiently maps the operations based on 32 registers for the source instruction to the 7 registers on the target processor. The optimization stage determines which of the 32 registers are used and in what sequence  
20 they are used in the instruction flow. Because it is rare that all 32 registers are used at any given time, the order of the registers that are used (e.g., 10) are determined and the registers on the target processor are temporarily assigned to those 10 registers. Given that an operation could only have a total of 3 different registers in active use at any given

time (e.g., the operation  $A+B=C$  requires two registers for A and B and a destination register for C), usage of the 7 available registers at the target processor is juggled. The mapping of the 7 target registers to the 32 source registers are constantly switched depending on the need dictated by the operation. Using the same example, if all the 7 registers are occupied and 2 of them are active for A and B, one of the other 5 registers is flushed into memory, and C is loaded into that flushed register. If the data that was flushed is needed again, it can be brought back to the registers by flushing one of the other registers. The optimization stage thus determines the optimized flow of information based on the characteristics of the target processor. Such information is passed onto the subsequent encoding stage 16. This provides the encoding stage the information about the 7 registers, and which to use to create the target instructions not only through actual functionality but to do all the swapping and flushing/purging of the registers. The optimization stage tells the encoding stage which of the registers to map and which to map out, etc.

The optimization stage also optimizes register mapping. Consider  $C=A+B$ , where A, B and C are already in memory. The source instruction for the R-3000 processor would require that register A and register B must be loaded from memory, register A is added to register B, and the result is in register C. The operation may require the purging of one or more of the registers on the processor to accommodate the variables, and the storing into memory the result C. The optimization stage recognizes from the instruction stream that the result of the addition is to be placed in register C, so the optimization stage outputs an instruction flow sequence which includes mapping the memory C to the register C, moving of A into register C and adding register B into register C, without fetching the

existing value of C from memory because the existing value of C will be overridden eventually. This process is conducted with processor registers and the value of register C does not need to be stored into memory until it needs to be flushed. It has saved the fetching of C from memory and the storing of C back out to memory until it is required,  
5 thereby reducing the overall memory load.

The above optimization process would not be applicable to  $C=C+A$ , since C must be loaded from memory into register because C is a source for the addition operation. The optimization stage recognizes the difference between the  $C=A+B$  and  $C=C+A$  scenarios and handles the optimization accordingly. For  $C=C+A$ , C is loaded from memory and A  
10 is added to register C. Without optimization as the instruction code for the R-3000 processor, the instruction sequence includes fetching A and C, adding and placing the result in a temporary register, and then moving the value in the temporary register into register C.

Referring to the instruction sequences  $C=A+B$  and  $C=M+Q$  described in connection  
15 with the decoding stage, instead of removing the overridden  $C=A+B$  instruction at the decoding stage, the dynamic compiler of the present invention may be configured to have the optimization stage to create an optimized instruction flow stream which omits the instructions corresponding to the  $C=A+B$  operation, thereby saving potential register purging and reload as well as reducing the amount of target code output.

20 Many other optimization rules may be developed for other operational scenarios and specific target processors without departing from the scope and spirit of the present invention. For example, optimization may include interleaving instructions for a target processor (e.g., SH4) which has a pipeline delay in which the data read from memory is

not useable until two instructions later. The optimization stage can optimize the flow of information by taking advantage of such delay to read other data before the previously read data is useable.

## 5 Encoding Stage

The encoding stage 16 encodes and optimizes the operation codes specific for the target processor. Using the examples of R-3000 and 80X86, the instructions for R-3000 are always 32-bit long (or 4 bytes). For the 80X86, the instructions are variable length, so the instructions might have a 1-byte instruction, a 3-byte instruction, 7-byte instruction and so on. Hence, it is advantages to use the smallest number of instructions and the instructions that take the least number of clock cycles to execute. Conceivably, one can have an instruction that is relatively long, say a 7-byte instruction, but is faster to execute such instruction than executing a number of instructions that are each 2-byte long. Based on the optimized flow of information determined by the optimization stage 14, the encoding stage 16 determines the optimized operation codes to achieve results equivalent to the source instructions.

In accordance with the present invention, the encoding stage provides several operation code options (cases) for achieving the same results. The particular case that is optimal depends on the data flow and intended result determined by the earlier stages. For example, for an addition, there may be four or more cases to achieve the intended addition, including increment by 1, decrement by 1, using the short range -128 to +127, and if outside of the range, apply a regular addition operation. To illustrate with an

example, to add 128 to a number, the encoding stage would optimize by subtracting (-128) instead, as  $-(-128) = +128$ . This is more efficient than using the normal approach of adding 128, because -128 is a short range number that is already available. Accordingly, the result of an addition operation is optimally achieved by a different physical operation, a subtraction operation in the example. Again, the present invention effectively interpolates the optimal operations to achieve the same results intended by the source instructions. In a way, the encoding stage optimizes the target operation code in a context sensitive way.

Similar and analogous logic may be applied to obtain other equivalent results to subtraction, division, multiplication, comparison, etc. without departing from the scope and spirit of the present invention. For example, for multiplication by 2, the number is shifted bit-wise in an operation in accordance with the R-3000 platform. To achieve a similar result of this shift variable operation, the encoding stage of the present invention may provide for several cases for shifting the number to be multiplied by 2 bit-wise to achieve the same result, depending which case is more efficient to obtain the equivalent result.

\* \* \*

The process and system of the present invention has been described above in terms of functional modules in block diagram format. It is understood that unless otherwise stated to the contrary herein, one or more functions may be integrated in a single physical device or a software module in a software product, or a function may be implemented in

separate physical devices or software modules, without departing from the scope and spirit of the present invention.

It is appreciated that detailed discussion of the actual implementation of each module is not necessary for an enabling understanding of the invention. The actual  
5 implementation is well within the routine skill of a programmer and system engineer, given the disclosure herein of the system attributes, functionality and inter-relationship of the various functional modules in the system. A person skilled in the art, applying ordinary skill can practice the present invention without undue experimentation.

While the invention has been described with respect to the described embodiments in  
10 accordance therewith, it will be apparent to those skilled in the art that various modifications and improvements may be made without departing from the scope and spirit of the invention. For example, the dynamic recompiler in accordance with the present invention may be implemented in computer hardware and/or firmware. Further, the dynamic recompilation concept may be applicable to translation of software  
15 instructions bases on other types of information handling systems, hardware and/or software platforms (e.g., Motorola 680X0, Intel 80X86, PowerPC-based systems).

Accordingly, it is to be understood that the invention is not to be limited by the specific illustrated embodiments, but only by the scope of the appended claims.